COMP6700/2140 Maps and Hashtables

Alexei B Khorev and Josh Milthorpe

Research School of Computer Science, ANU

May 2017

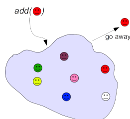
Topics

1. *HashMap* and *HashSet* implementing classes
2. Hash Table — an implementation Data Structure
3. Good Hash Code
4. Ergodicity — maximising the use of a value set

Set and *SortedSet*

The extensions of *Collection* type which disallow duplicates (identical elements): a repeated invocation of `add(elem)` with the same element `elem` (or such `elem1`, that `elem.equals(elem1)` returns `true`) returns `false`, and the collection remains unchanged. *Set* types, therefore, are sets in the mathematical sense (some computer scientists call them “bags”). The elements of a set are unordered. The subtype *SortedSet* (extension of the *Set* interface) represents a collection, whose elements can be *compared* — they implement *Comparable* interface. The ordering allows to introduce additional methods:

- `comparator()` — returns the comparator used with this sorted set (`null` in case of *natural ordering*)
- `first()`, `last()` — smallest and largest elements
- `subset(eMin,eMax)`, `headSet(eMax)`, `tailSet(eMin)` — subsets of elements in given ranges

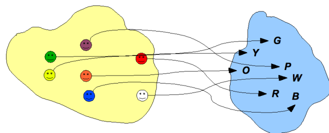


Set is implemented by the *HashSet* class with a hash table as the DS. Content modification (`add`, `set`) and testing (`contains`) are $\mathcal{O}(1)$ operations. *SortedSet* is implemented by the *TreeSet* class with a binary tree as the DS. If the implementation can maintain the *balanced tree*, the search and modify operations are $\mathcal{O}(\log_2 N)$. Example — SetTest.java.

Map and *SortedMap*

Unlike all previous types from JCF, the *Map* family of types does not extends the *Collection* interface: their contract is different — to represent not a collection of elements, but a *correspondence between two collections*. So, a map contains **key/value pairs**, with *no duplicate keys* (the keys form a set) and *at most one value* for each key. *Map* is a model of a mathematical abstraction called *function*. The interface Map<K,V> operations are:

- `V put(K key, V value)`
- `V get(K key)` and `V remove(K key)`
- `boolean containsKey(K key)`
- `boolean containsValue(V value)`
- `public Set keySet()` — returns a **view** of the set of keys
- `public Collection values()` — returns a **view** of the collection of stored values

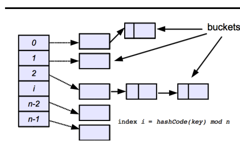


Map of Smiley to Character

The *SortedMap* extension requires the set of keys be sorted. Methods like `firstKey()` and `lastKey()` are added. The *Map* interface has two major implementation classes (similar to *Set*) — *HashMap* which uses a hash table DS for the implementation (with similar $\mathcal{O}(1)$ performance for put/get operations), and *TreeMap* which implements *SortedMap* in the similar to *TreeSet* way (with $\mathcal{O}(\log_2 N)$ efficiency). Example — MapTest.java.

“The Great Invention of Computer Science”

An ordinary array can be thought of as a table or mapping between an integer index value and a data value. Arrays enable efficient random access to the data, $\mathcal{O}(1)$. The performance worsens when we copy in more than the original array can take — can we retain good performance in this situation? Also, often one wants to index entries *not* by integers (by *String*y names) — can we do it with $\mathcal{O}(1)$ efficiency (instead of resorting to linked lists)?



A data structure which allows us to do such things is a *hash table*. A HT is an *array of linked lists* **plus** a *hash code function*, which maps the data element to the index value of the array.

- There is no specific ordering of data in a hash table.
- The `hashcode` mapping can be many-to-one.

We expect that different keys compute to different table indices, but if not — *hash collision* — we add a new entry to the bucket (list before Java 8, Red-Black tree after). One should keep the HT balanced and avoid growing some bucket lists too long. The performance of the data entry/retrieval operations is $\mathcal{O}(1)$ (for well-balanced HT) and $\mathcal{O}(N)$ (for opposite extreme).

Balancing a hash table: hashCode()

To achieve the optimal performance (reduce the number of collisions), one should:

- specify the initial bucket count (size) of a hash table sufficiently high (150% of the expected number of elements)
- if a hash table gets rather full (few empty buckets), it needs to be *rehashed* — a table with more buckets is created and all elements are inserted into this new table (in JCF's *HashMap*, this is done automatically when the number of elements exceeds a load factor, 75% of the table size; rehashed table is twice bigger)
- to implement the `hashCode()` function ***properly*** for the class of elements which a going to be hashed — only two elements which are equal, `e1.equals(e2) == true`, must have he same value `e1.hashCode() == e2.hashCode()`. Every class which redefines `equals()` method should also redefine `hashCode()`.

Here is an example of the `hashCode()` for a simple *Item* class:

```
class Item {
   private String description;
   private int partNumber;
      ...
   public int hashCode() {
      // note use of prime number multipliers
      return 13*description.hashCode() + 17*partNumber;
   }
}
```

How to write the `hashCode()` function

Hash codes are meant to generate array indices which are *uniformly and randomly* distributed over the size of the array. For this reason they work on an idea of "scrambling up" the values of the data being hashed, so that it is not possible to predict where a particular object will be stored. This is how `hashCode()` method for the *String* class might have been implemented:

$$hashCode(s) = \sum_{i=0}^{n-1} s_i \cdot 31^{(n-1-i)},$$

where `n = s.length()` and s_i is the code of the character `s.charAt(i)` (Exercise: implement this function as a Java method.) *Very similar* inputs evaluate to *quite different* outputs:

String	Hash value	String	Hash value
`Hello`	69609650	`Harry`	69496448
`hello`	99162322	`Hacker`	-2141031506

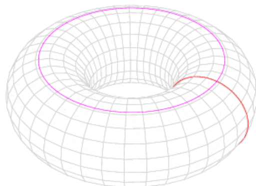
To define a hash function for a class whose instances can be stored in HT is a subtle task. Standard API's Java classes should have good hash codes already. When you override `hashCode()` for your own type, do it using the principle exemplified on the previous slide's *Item*, and use two- (or more) digit mutual primes (`13, 17, 31`...) as factors.

Nice Physics Analogy — *Ergodicity*

For simplicity, let's assume that we need to define the hash code function for a newly defined type which contains two fields of existing types (with well defined hash codes):

```
class NewClass {
    OldClassOne one;
    OldClassTwo two;

    public int hashCode() {
        return m1 * one.hashCode() +
                   m2 * two.hashCode(;
    }
}
```



Why did we choose the multipliers `m1, m2` to be mutually prime integers? Such choice allows us *the set of hash code return values* to be *closer* to the *full set* of all possible values which the two constituent hash codes *can take*; mathematicians call it *a tensor product*. Since every hash code is calculated `mod N`, the values of `hashCode()` of *OldClassOne* and the values of `hashCode()` of *OldClassTwo* cover a circle, and their "tensor product value set" is a *torus*. The value set of `NewClass.hashCode()` looks like a winding *closed curve* of the torus surface. The more windings it does around the two base circles, the denser it covers the torus, and the more points it will contain. These *winding numbers* (up to unessential common factor) are those multipliers `m1` and `m2`.

Where to look for this topic in the textbook?

- Hortsmann's Core Java for the Impatient, Ch. 4.2.3, 7.3, 7.4